IMPLEMENTATION OF AN ADAPTIVE SCHEDULING ALGORITHM FOR THE MUNIX OPERATING SYSTEM

Ronald Edward Joy

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

Implementation of
An Adaptive Scheduling Algorithm
for the
MUNIX Operating System

by

Ronald Edward Joy

December 1975

Thesis Advisor: Gerald L. Larksdale, Jr.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
Implementation of
An Adaptive Scheduling Algorithm
for the
MUNIX Operating System

5. TYPE OF REPORT & PERIOD COVERED
Master's Thesis
December 1975

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)
Ronald Edward Joy

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Naval Postgraduate School
Monterey, California 93940

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Naval Postgraduate School
Monterey, California 93940

12. REPORT DATE
December 1975

13. NUMBER OF PAGES
65

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)
Naval Postgraduate School
Monterey, California 93940

15. SECURITY CLASS. (of this report)
Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)
Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
MUNIX, UNIX, Adaptive Scheduling Algorithm, Operating System

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
The design, implementation, and evaluation of an adaptive scheduling algorithm for the MUNIX operating system is reported here. MUNIX, a multiprocessing version of UNIX, was designed to run on a dual PDP 11/45 multiprocessor system. Topics covered include: a survey of adaptive scheduling, laboratory equipment configuration, scheduling with MUNIX, benchmark testing, and non-adaptive scheduling changes.

DD FORM 1 JAN 73 1473 (Page 1)
EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Conclusions and suggestions for possible improvements are also included.

Implementation of An Adaptive Scheduling Algorithm for the MUNIX Operating System

by

Ronald Edward Joy
Captain, United States Air Force
B.S., United States Air Force Academy, 1971

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1975

ABSTRACT

The design, implementation, and evaluation of an adaptive scheduling algorithm for the MUNIX operating system is reported here. MUNIX, a multiprocessing version of UNIX, was designed to run on a dual PDP 11/45 multiprocessor system. Topics covered include: a survey of adaptive scheduling, laboratory equipment configuration, scheduling with MUNIX, benchmark testing, and non-adaptive scheduling changes. Conclusions and suggestions for possible improvements are also included.

TABLE OF CONTENTS

List of Tables

List of Drawings

Acknowledgement

Several individuals made the process of performing research enjoyable. To mention a few - my wife, Patty; my advisor, Gerry Barksdale; and my second reader, Bill Allen. A special note of recognition is due Bill. He assisted me in starting on the non-adaptive scheduling changes. The NPROC change (see section II.E.2.) had his close supervision. To be more precise, I assisted him in making and debugging the change.

I. INTRODUCTION

In the fall of 1974, the Computer Science Group at the Naval Postgraduate School acquired a fairly large amount of computer hardware and a limited amount of software. The intent of acquistion was to integrate the hardware and software to support signal processing research. The hardware consisted of two PDP 11/50 computers made by Digital Equipment Corporation, one CSP 30 processor made by Computer Signal Processors Incorporated, and various associated peripherals described in section II.B.2 and section II.B.3 (see figure 3).

An agreement with Bell Laboratories provided the software consisting of an operating system called UNIX [15]. UNIX, as delivered, did not have the capability to fully utilize all the system resources necessary to support signal processing. As a result, several research projects were done in this area. MUNIX [7], a multiprocessing version of UNIX, was one of the projects done. Note that where the word MUNIX is used in this thesis, UNIX may be substituted. The changes made to MUNIX may easily be incorporated into UNIX.

One of the goals of the computer system was to handle real-time, timeshared, and batch processing [12]. It was found that the scheduling algorithm used in UNIX could be improved for the equipment configuration being used. An

excessive amount of time was being wasted swapping users in and out of core. This wasted time was a function of both the scheduling algorithm and the amount of available core. Figure 1 shows the amount of real time, in minutes and seconds, required to run the same benchmark program (see Appendix B) with different amounts of core available.

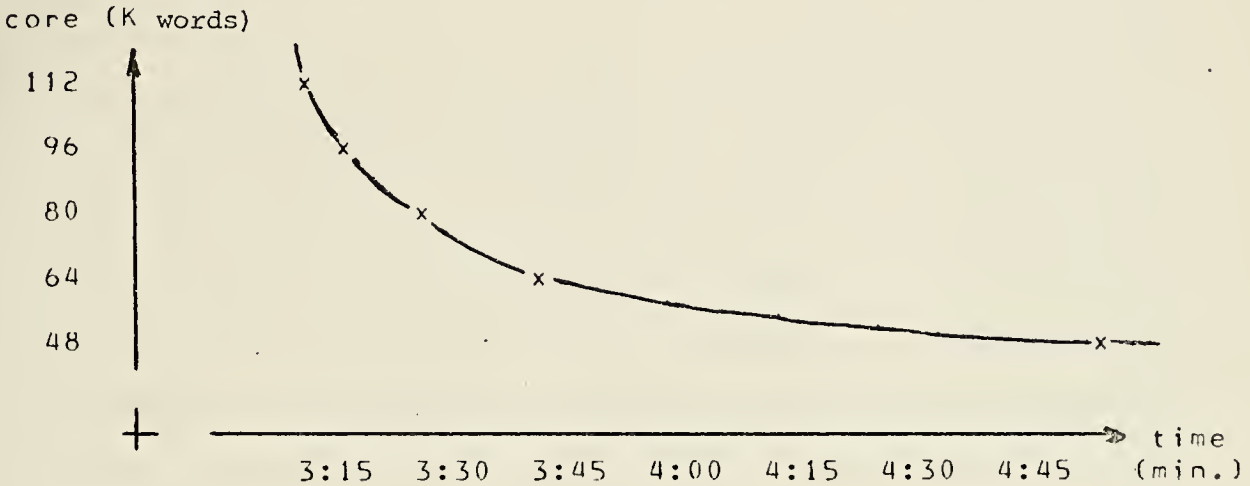


Figure 1. Benchmark Real Time Vs. Core Size

Implementing a scheduling algorithm that gave the interactive user faster response times and increased throughput was desired. Since a member of the Computer Science Group was interested in adaptive scheduling, thesis research was accomplished in this area and reported here.

II. BACKGROUND INFORMATION

A. ADAPTIVE SCHEDULING - A SURVEY

1. General

Scheduling algorithms can be placed into three basic classifications - round robin, priority, and dynamic or adaptive control. Algorithms based on round robin and priority assignment techniques have a common characteristic: the processor is switched from the process currently being serviced to a new process at the end of a fixed time quantum or when a new process is of higher priority. This switching usually has a significant overhead and reduces system utilization.

When an operating system is getting close to saturation, the round robin scheduling algorithm often fails to give an adequate response time to the time-sharing user [16]. With a priority type algorithm, processes are assigned priorities as they are entered into the system. The user supplies the information necessary for the operating system to assign a priority to the process. The information supplied can consist of an estimate of CPU time, an estimate of the amount of core required, and the number of

input/output devices required. This information usually is an estimate of the maximum time or the maximum amount of primary memory required.

Adaptive control solves the problems of the round robin and priority scheduling algorithms by giving adequate turn-around times to all processes except those which are run in the background, that is, processes not in contention for immediate service. Several papers illustrating adaptive control scheduling techniques are discussed below.

Northouse and Fu [13] develop a scheduling algorithm based on adaptive control and clustering techniques. Bernstein and Sharp [2] and Sharp and Roberts [16] develop an algorithm based on the principle, "don't do anything unless you have to." This avoids the system overhead of process-switching and swapping as much as possible.

2. Adaptive-Control and Clustering Technique

An adaptive controller can be referred to as a closed-loop or a feedback type system with the characteristics of the controlled process. Northouse and Fu [13] proposed their batch scheduler as an adaptive controller with three basic units: a classifier, a performance evaluator, and a distributor.

The classifier made an "a priori" classification of all incoming jobs based on information supplied by the user on

his job card. A clustering algorithm was used to establish clusters. The parameters used by the clustering algorithm were:

1) CPU time used.
2) number of tape drives.
3) number of input cards.
4) programming language.
5) number of drum or disk files.
6) number of output pages.

Much effort went into the proper classification of clusters with the following final result:

cluster I-medium CPU, large tape file jobs
cluster II-large jobs
cluster III-small jobs
cluster IV-medium CPU, small tape file jobs

The performance evaluator monitored the system performance in specific areas and compared these evaluations to desired responses. Specific areas monitored included central processor utilization, printer traffic, drum and/or disk traffic, and tape drive utilization. If efficiency in a monitored area dropped below a minimum acceptance level a change in the job stream was made.

A performance index was calculated and attempts were made to optimize this index for the next subinterval (variable lengths). The job stream was then determined using this index and a linear programming technique. The distributor implemented the job stream that was calculated by the performance evaluator.

As jobs were executed, their statistics were used by

two more components called the data collector and the data base updater. The updater made the system a closed loop system by continually updating the data base from which the linear programming function made its decisions.

Northouse and Fu, after running two simulations on their scheduler, concluded:

1) The scheduler was able to adapt to changing work loads.

2) The job stream had definite clusters.

3) The programming language was an important parameter for classifying clusters.

4) The distributor required few calculations and was easily updated.

5) Selected clusters could be forced through the system reducing their turnaround time.

6) Using a proper selection policy had a significant impact on decreasing turnaround time.

3. Adaptive Policy Driven Scheduler

A "Policy-Driven Scheduler" attempts to deliver computational resources at a rate determined by some type of criterion or policy function. Berstein and Sharp [2] define their policy functions in terms of "resource units" and "age of interaction." An interaction consists of: a request from a user to the system, some system service, and a reply from

the system back to the user. An edit command on a time-sharing system is an example of an interaction. The execution of a batch job is another example.

The age of the interaction at some time "t" is the elapsed time from when the user made the request to the system until time "t". Resource units are a measure of service received by a particular interaction. Wi is an arbitrary non-negative time cost for the i-th resource and Rij(t) is the number of units of the i-th resource used by the j-th interaction at age "t". The total resource units of the j-th interaction at age "t", Rj(t), is equal to the sum of all the Wi times Rij(t).

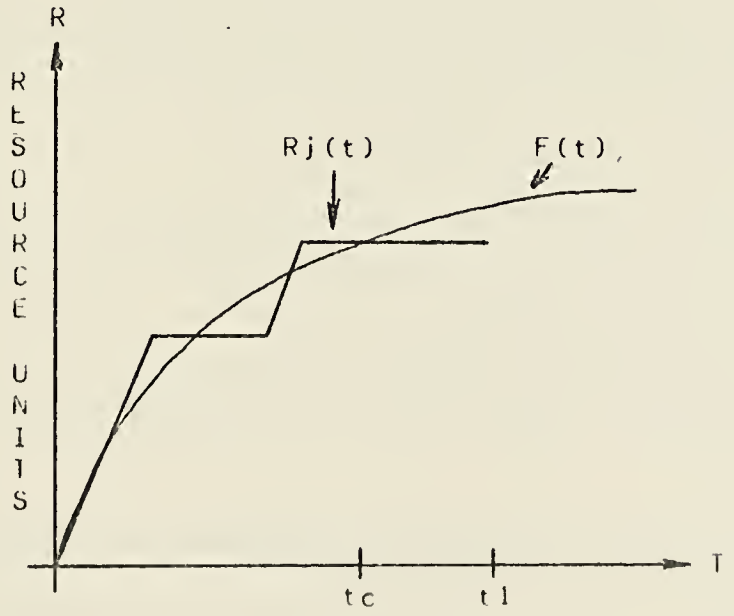


Figure 2. Resource Count and Policy Function vs Age.

The resource count is a nondecreasing function of time. Figure 2 shows the resource count function for the j-th interaction up to age t1. A slope of zero indicates periods of no service while a positive slope indicates periods of resource consumption. The policy function, F(t), is shown as a curve.

The goal of the scheduling algorithm is to keep the terminal point of each interaction above the policy function. The total amount of resources required to complete an interaction is not usually known in advance. Thus, the algorithm tries to maintain Rij(t) greater than or equal to F(t).

An interaction is critical at time "t" if Rj(t) is less than F(t). Interactions are ordered according to a measure called "critical time." Critical time is defined as:

$$t0 + tc - t1,$$

where t0 is the current time and t1 is the current age of the interaction. tc is the last age of the interaction at which time it went critical. Figure 2 shows an interaction of age t1 which became critical at age tc and which is still critical.

Critical time changes only when service is received and thus only needs to be updated at that time. This property insures that the queues of processes ordered by this quantity remain ordered as time progresses. After a process receives service it must be relocated in the queues.

The scheduler has two queues. The "core queue" is a linked list of processes in main memory and the "drum queue" is a linked list of processes not in main memory. Both queues are ordered by critical time. Processes from the head of the "drum queue" are transferred into main memory if room is available. If room is not available for a process, a swapping decision is made based on a comparison of the critical times of the first process on the drum queue and the last process on the core queue.

This scheduler reduces overhead caused by unnecessary swapping by prohibiting the replacement of a process in core by a noncritical process which is not in core. The rules that make up the swapping decision are:

1) A critical process in core is not eligible to be swapped out (designed to prevent thrashing).

2) Processes which are inactive because they are awaiting communication from a terminal are given a critical time of t(e).

3) A noncritical process which is not in main memory will be swapped in if room exists or if room can be created by swapping out processes with a critical time of t(e).

4) A critical process which is not in main memory will displace a noncritical process which is in main memory.

Bernstein and Sharp were unable to detect thrashing using these constraints.

The most critical process which is in main memory and

ready to execute is given the processor. The period of use is one time quantum or until the process voluntarily relenquishes it, which ever occurs first. The processor is again dispatched after a swapping decision is considered.

The policy function controls the service received by each interaction. Static policy functions must be set conservatively to avoid response problems during heavy loads; however, it was found that during light load periods these conservative settings resulted in a wide range of service rates to similar jobs. Sharp and Roberts [16] found that varying the policy functions as the job load changed greatly reduced the service variance.

Bernstein and Sharp showed that their policy driven scheduler was far better than a round robin scheduler in terms of "internal response time" measured in seconds:

SCHEDULER	MINIMUM	MEDIAN	MAXIMUM
Round Robin	4.5	10.8	102.4
Policy-Driven	0.5	1.7	3.8

Table I. Round Robin Vs Policy-Driven Scheduler

Sharp and Roberts demonstrated that their adaptive policy driven scheduler was far better than the static policy driven scheduler with the following results:

SCHEDULER	RESPONSE TIME	CPU UTILIZATION	DISC UTILIZATION
Non-Adaptive	5.1 sec.	61%	40%
Adaptive	1.7 sec.	66%	44%

Table II. Non-Adaptive Vs Adaptive Scheduler

Note the differences in the magnitude of the response times in both comparisons.

An adaptive policy driven scheduler as it pertains to the MUNIX operating system for the PDP-11/50 will be discussed in detail in Chapter III.

B. LABORATORY EQUIPMENT CONFIGURATION

1. General

Although MUNIX is a multiprocessor operating system, all testing was done with only one processor active. This was done to control testing. Documentation of the laboratory equipment configuration is necessary because test results depend on which of the two systems is used. Figure 3 shows the laboratory equipment and configuration. During the design and implementation of the adaptive scheduler, the operational equipment consisted of two PDP 11/50 CPU's (labeled A and B) with the following equipment:

2. System A

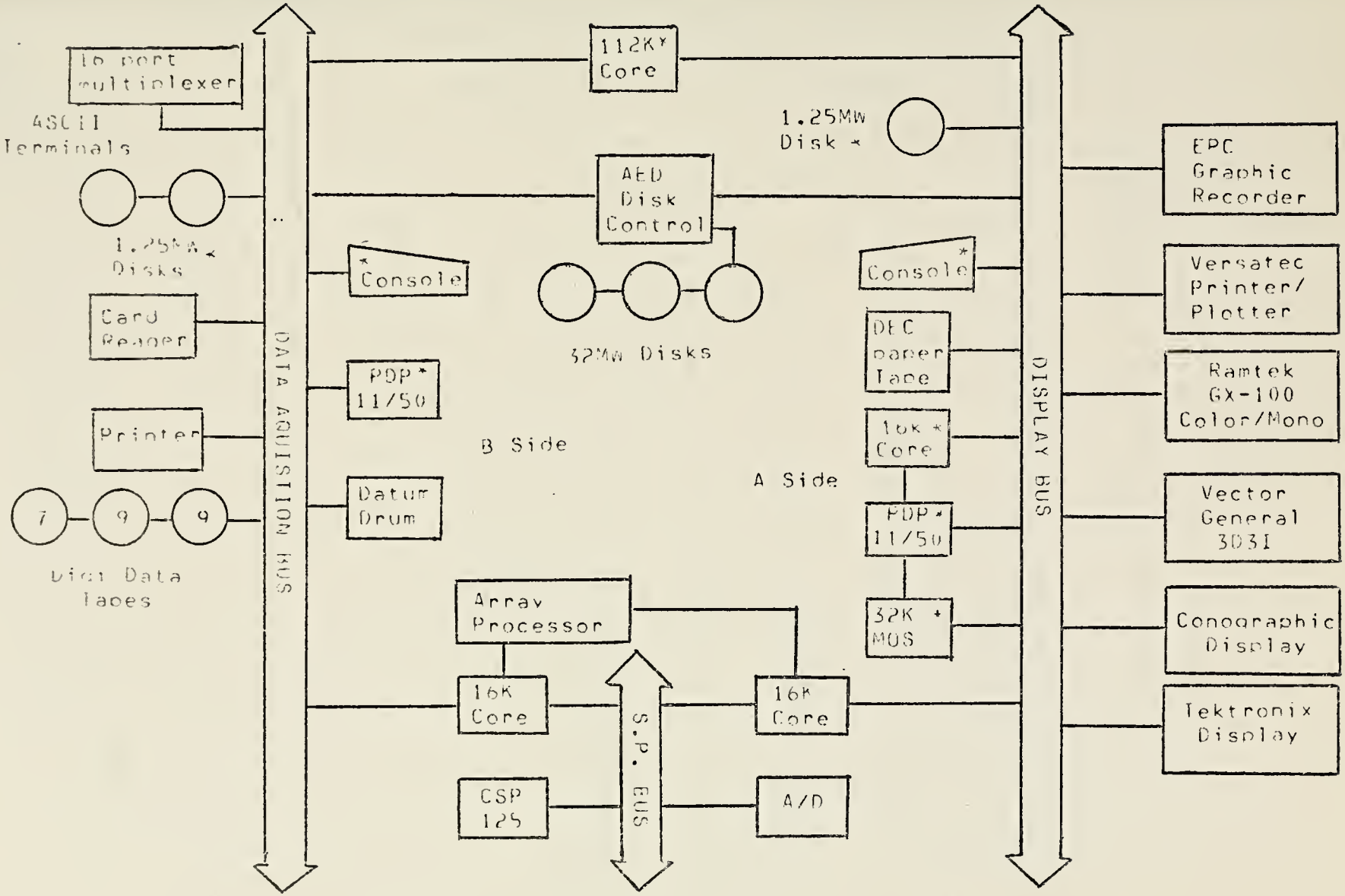
32K MOS memory (450 nsec. access time)
16K core memory (750 nsec. access time)
16K CSPI memory (900 nsec. access time)
1 DEC LA30-C terminal
1 disk cartridge (RK05 equivalent)
1 Versatec printer/plotter
1 paper tape reader/punch
1 Vector General 3D3I vector display terminal
1 Ramtek raster scan color display unit
1 Tektronix 4014 display terminal
1 Hughes Conographic console
1 Data Tablet
1 EPC graphic recorder

3. System B

96K CMI core (850 nsec. access time)
16K CSPI core (900 nsec. access time)
1 DEC LA30-C terminal
2 DECpack disk cartridges (RK05)
1 DEC DH11-AC communications multiplexor connected to (up to 16) remote terminals
1 card reader (600 cards/min)
1 impact printer (400 lines/min)
2 nine track magnetic tape drives
1 seven track magnetic tape drive

4. System A and B Differences

The most important difference between the two systems is the amount of memory available. System B has more memory than system A and therefore can have more processes in resident core. This has a significant effect on any benchmark testing (see section II.D.). In addition, the speed of the memory must also be considered.



* Components Active During Benchmark Testing

Figure 3. Laboratory Equipment Configuration

C. SCHEDULING WITH MUNIX ON THE PDP-11/50

MUNIX [7] is a multiprocessing version of UNIX [15], a general purpose, multiuser, interactive operating system usable on the Digital Equipment Corporation PDP-11/40, PDP-11/45, and PDP-11/50 computers. UNIX was developed by Bell Laboratories.

In order to understand the MUNIX scheduling algorithm and implement a new one it was necessary to learn C, a high level programming language. Several references were helpful in this endeavor [1,10,18].

Although parts of the MUNIX operating system have been documented [7,12], Appendix A will attempt to completely document the portions related to scheduling.

D. BENCHMARK TESTING

Benchmark testing is often used to evaluate and compare the performance of one computer system relative to another. A benchmark program was written to test scheduling algorithms. It was necessary to use processes that accomplished input, output, computations, and compilations. A discussion of the benchmark program used may be found in Appendix B.

E. NON-ADAPTIVE SCHEDULING CHANGES

1. General

After studying the scheduling algorithm used by MUNIX, it was decided to make some non-adaptive changes before implementing an adaptive scheduler. Three areas were modified to make the algorithm more efficient and have a better basis to start performance evaluations on the adaptive scheduler. The changes are documented in detail in Appendix C and summarized here.

2. Maximum Number of Processes (NPROC)

a. Change

NPROC was a static upper limit for the number of processes in the process table. The static upper limit was replaced by a dynamic one, thus saving process table search time.

b. Evaluation

The benchmark program (see Appendix B) was run against the scheduling algorithm before and after this change. Four runs were made, two with a drum being used for

/TMP files (temporary files), and two without. The results are listed in tables III and IV. Real, user, and system times are shown in minutes and seconds. Appendix B explains how the system calculates these times and estimates their accuracy. This testing was accomplished on the "B" system (see section II.B.3.).

BEFORE CHANGES		AFTER CHANGES	
real	6:02	real	6:00
user	2:12	user	2:00
sys	:42	sys	:41

Table III. NPROC Change Evaluation with No Drum

Before Changes		After Changes	
real	4:49	real	4:38
user	1:58	user	1:48
sys	:41	sys	:42

Table IV. NPROC Change Evaluation with Drum

3. Looping in Function Sched

a. Change

As described in section B.1.b and B.1.c.(2) of Appendix A, two loops in sched were shortened so no

unnecessary code was executed.

b. Evaluation

The benchmark program (see appendix B) was run before and after the changes. Several runs were made because the statistics showed no significant savings (see Table V). Testing was accomplished on the "A" system.

Before Changes	After Changes
real 7:08	real 7:08
user :46.6	user :46.6
sys :18.0	sys :17.8

Table V. Looping Change Evaluation

4. Size Check

a. Change

Function sched was changed to make an additional check before swapping a process out of core. The size of the incoming process had to be smaller than or equal to the size of the outgoing process. If the incoming process is larger than all processes eligible for swapping, no size check is made. This task is accomplished using two passes. See Appendix C for a detailed explanation.

b. Evaluation

Several runs were made with the benchmark program on the "A" system because of the 26 percent savings realized in real time (see Table VI). This change was also tested on the "B" system, but a savings of only 6 percent was found there. The difference in savings is explained by the significant difference in available memory (the "B" system has three times as much user space) for each system.

BEFORE CHANGES	AFTER CHANGES
real 7:08	real 5:18
user :46.6	user :45.3
sys :18.0	sys :17.9

Table VI. Size Check Change Evaluation

III. ADAPTIVE SCHEDULER

A. DESIGN

The adaptive scheduler described in section II.A.3 was implemented with a few minor changes.

1. Goals

There were two goals that the adaptive scheduler attempted meet.

a. Improve system throughput by reducing the amount of process swapping (in and out of core).

b. Give the interactive user better response time. The MUNIX scheduler basically gave users a round robin type of service.

2. Design Changes

Sharp and Roberts [16] measured the criticallity of a process as the time from where the process last went critical until the current time (see figure 2). The implemented scheduler measures the critical time as the

vertical distance from the policy function to the resource count. This design change was made to facilitate a more efficient calculation of priorities. Sharp and Roberts calculated priorities on a fixed period basis. In the current scheme priorities are calculated whenever they are needed. There are two reasons for this change:

1) The policy function is changed whenever the job (process) load reaches predetermined amounts. When the function is changed, all jobs, both in and out of core, have to have their priorities recalculated with the new policy function.

2) Depending on the fixed time period, jobs may receive an excessive or insufficient amount of resources.

By recalculating the priorities on a continuing basis, no special software is needed to recalculate the priorities after policy functions have been changed. Also, everytime a scheduling decision is made, it is made with the latest priorities of all the jobs concerned.

B. IMPLEMENTATION

1. Process Table Control Variables

a. ptime - was changed from a character variable (maximum value of 127) to an integer variable

(maximum value of 32767). The use of the variable was also changed (see Appendix A section A.1.f). Currently it is used as a counter for the total number of seconds since a process (job) has last had any teletype input. It is also used to calculate the priority of the process.

b. p←flag - was changed from a character variable to an integer variable because two additional bits were needed as special indicators.

1) PSTM - (value of 400 octal) when set means that the process has received a minimum amount of resources in a minimum amount of time and from this point on will be run strictly as a background process.

2) TPWAIT - (value of 1000 octal) when set means that this process is waiting for terminal input and will not be scheduled to run again until terminal input has been made.

c. p←resr - was added as an integer variable. It is used to keep track of the amount of resources received by the individual process. The U-vector (see Appendix A section A.2) of each process has two variables, u←utime and u←stime, that contain the same information. These variables could not be used for two reasons:

1) The U-vector is in core only when the

process is core resident. Priorities must be calculated both when the process is in and out of core.

2) Because of the inter-dependency of processes [15] (page 370), the parent (or grandparent) process accumulates the resource units (u.u_utime and u.u_stime) of its childern (or grandchildern).

This new variable only accumulates resource units for the process it is related to. p.p_resr is incremented in program clock.c in the same places as u.u_utime and u.u_stime.

2. Priority Calculations

Subroutine "schpri.c", schedule priority, was written to calculate process priorities (see Appendix D). There were two possible areas of the operating system that the priorities could be calculated.

a. Subroutine swtch, in program slp.c, is entered each time the operating system changes (switches) from one process to another.

b. Subroutine sched, also in program slp.c, is entered each time a process is being considered for swapping.

A test was run to examine the number of times the two

subroutines are executed relative to each other. It was found that swtch is executed approximately thirteen times as often as sched. Sched also decides which process should be core resident while switch decides which process should be executed next. As a result of these two observations, sched was chosen as the place to calculate priorities.

3. Scheduling Algorithm

Subroutine sched (see Appendix A section B.1) is described with adaptive changes. This subroutine is a process with an infinite loop initiated from function main. Sched is executed, put to sleep, awakened, and executed again. Sched's main job is to swap processes in and out of core. It accomplishes this task using the following algorithm:

a. Set the first/second pass indicator (fspass) to a value of first pass.

b. If there are any processes in the swap file, find the one that is the most critical and try to transfer it into core. If the swap file is empty, then go to sleep on the RUNOUT flag which indicates there are no processes in the swap file. When awakened go to "b." and continue. Sched may try up to a maximum of three different ways to bring this process into core.

c. If room exists in core then transfer the process into core.

d. If room does not exists in core then sched looks for what it calls "easy core". Easy core is core that belongs to a process that is not a system process, not locked, and waiting for some type of input or output. An additional constraint that applies only on the first pass is that the size of core needed for the incoming process is less than or equal to the size of core of the outgoing process. If easy core is found, then the outgoing process is swapped out and sched goes to "c." to continue. Sched repeats this until either the process can be transferred in, or no easy core is available.

e. If no easy core is available, then sched searches all the processes that are in core for the one that has the lowest priority and meets the following constraints: the process must not be a system process, not locked, sleeping, and not currently being run on the other processor. Two additional constraints, that are made on the first pass only, are the size check mentioned in "d." above and a check that the process can be ready to run instead of sleeping. At this point there are two possible states to consider.

1) The lowest priority process eligible to be swapped out is critical. If this is the first pass,

change the first/second pass indicator to second pass and go to "d." , or if this is the second pass and no processes are eligible to be transferred out, go to sleep on the RUNIN flag (processes are in the swap file). When awakened go to "a." and continue.

2) If the lowest priority process eligible to be swapped out is not critical or this is the second pass and a process meets the requirements in "e." above, then swap the indicated process out of core and go to "c." and continue.

Two additional versions of the above algorithm were tested, and will be discussed in section IV.A.

IV. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

1. Critical Processes

a. Change

Sharp and Roberts' algorithm [11] did not allow any process which was critical to be swapped out of core. That same constraint was implemented by changing B.3.e.1 and B.3.e.2 in section III as follows:

B.3.e.1 The lowest priority process eligible to be swapped out is critical. If this is the first pass, change the first/second pass indicator to second pass and go to "d.", or if this is the second pass, go to sleep on the RUNIN flag and when awakened go to "a." to continue.

B.3.e.2 If the lowest priority process eligible to be swapped out is not critical, then swap the process out and continue with "c." above.

This change was implemented by only searching for processes with a priority greater than zero.

b. Evaluation

Because of the inter-relationship between processes, this change was able to lock all of core and put the operating system into a deadlock condition. For example, consider the case where both the parent and the child processes are critical and the parent is waiting for the child's termination. The child cannot terminate because it is not in core and cannot get into core because the parent is critical (and cannot be swapped out).

2. Non-Critical Processes

a. Change

It was thought, that by not trying to transfer a non-critical process into core (by swapping another out), swapping time could be saved. This change was implemented by adding the following after III.B.3.e. above.

If the incoming process is not critical then go to sleep on the RUNIN flag. When awakened go to "a." above to continue.

b. Evaluation

Although this change did not cause a deadlock, it did create an unsatisfactory result. Example:

The parent is out of core and has two childern in core. The first child is a compute bound job and has received a lot of resource units. The parent becomes critical and forces the non-critical child (compute bound) out. The second child dies, the parent is now waiting on the first child, but he cannot get back into core until he is critical. So the computer has nothing to do until the child is able to get back into core by forcing the parent out and back into a new location.

3. Implemented Algorithm

a. Change

All the changes are explained in section III.B.3.

b. Evaluation

The benchmark program was run several times on the A processor with the results listed in Table VII.

BEFORE CHANGES		AFTER CHANGES	
real	5:18	real	8:33
user	:45.3	user	:45.9
sys	:17.9	sys	:18.8

Table VII. Implemented Algorithm Evaluation

It is obvious that the real time is slower. This can be explained in part by the amount of time spent swapping processes. The MUNIX scheduler solves this problem by requiring that a process stay in core at least three seconds and once out, stay out at least two seconds. A similar effect could be accomplished with the adaptive scheduler by changing the process' p←time. This was decided against because the adaptive scheduler would then be a modified (and probably less efficient) MUNIX scheduler.

4. Goals

The goals of this thesis were not met by the adaptive scheduler. However, they were met in part (non-adaptive scheduling changes - section II.E.) by the research accomplished while implementing the scheduler.

a. System thru-put was improved by reducing the amount of process swapping (see section II.E.4.).

b. The interactive user is not given a better response time, but all users are. This was accomplished by improving the efficiency of the current scheduling algorithm.

B. RECOMMENDATIONS

1. Adaptive Control

a. MUNIX

The adaptive control scheduling algorithm concept, when applied to MUNIX or UNIX will need more careful consideration. UNIX uses a hierarchical process structure which creates process inter-dependency problems. An example of the problem can be found in the "time sh benchmark" command sequence (Appendix B). The "time" command is the parent to the "sh" command, which is the parent to the "benchmark" command file. The "benchmark" command file will go two additional generations lower in all "C" compile commands. It is entirely possible to be eight or nine generations deep without executing any involved command sequences. Thus, it is a frequent occurrence that the currently active child will have numerous (intentionally) waiting ancestors which have no computational requirement until the child terminates. The failure of the present adaptive control effort seems to stem largely from the fact that each process was not considered on its own merit.

When a parent is waiting for a child to terminate, the parent should not be in competition for resource units with the child. There are two possible solutions to this problem:

1) Any process that is in the wait state should not have its "p←time" increased. This would not allow a process to change priority while it is waiting on another process. This change would require a "status" check where "p←time" is incremented.

2) If the parent process is waiting on a child, set a status flag so that the parent will not be put in contention with its non-terminated child. This change would be considered the general case, but it would require more software changes.

b. Other Operating Systems

Implementing an adaptive control scheduling algorithm with a minimal hierarchical structure seems to be straight forward. Sharp and Roberts [16] reported no serious problems with their implementation.

2. Additional Research

Additional research should be undertaken to analyze process interactions in UNIX. To do this, a comprehensive systems instrumentation package must be developed. In particular, a better timing mechanism, a "complete" resource utilization accounting system, and a selective event tracing capability are needed. In light of

the level of improvement Sharp and Roberts [16] report, additional research with adaptive control and MUNIX appears warranted.

APPENDIX A: PROCESSES AND SCHEDULING

A. PROCESS INFORMATION

Any time the word "process" is used in this appendix, the word "interaction" from section II may be substituted.

1. Process Table (proc.h)

This table contains the control information for process scheduling. Currently, the table may contain up to fifty active processes, each occupying a process block with thirteen data elements describing it. A process is assigned a process block when it is created and relinquishes it when it is deactivated. The elements and their meanings are:

a. p_stat - a process scheduling status with the following possible states:

(1) SSLEEP - this process has been put to sleep (not available to run).

(2) SWAIT - this process is waiting for some type of input/output completion.

(3) SRUN - this process is ready to run.

(4) SIDL - this process is active but not in any other status.

(5) SZOMB - this process has terminated but information in the process control block is required for other uses.

b. p_flag - process status of the memory manager with the following possible states:

(1) SLOAD - this process is loaded in main memory.

(2) SSYS - this process is a system process.

(3) SLOCK - this process should not be swapped out of main memory and is therefore locked in.

(4) SSWAP - this process is being swapped out of core.

(5) SMDF - this process must run on the first (B) processor.

(6) SMDS - this process must run on the second (A) processor.

(7) SANYP - flag used for processor masking.

(8) SBRKPT - system break point, used as a debug tool.

(9) SGOING - this process is currently being run by some processor.

c. p_pri - process priority. The priorities are whole numbers and range from -128 (highest) to 127 (lowest).

d. p_sig - process signal indicator.

e. p_uid - the unique id assigned to the user that created this process.

f. p_time - the total time in seconds that this process has been in or out of core.

g. p_ttyp - the id of the terminal associated with this process.

h. p_pid - the unique id assigned to this process.

i. p_ppid - the unique id assigned to the parent of this process.

j. p_addr - the address (memory or disk) of the first word of the process' "u vector" (described below).

k. p_size - the amount of non-shareable core this process needs.

l. p_wchan - holds a number that can be a channel address or a special indicator. The process is put to sleep or suspended with this number and it can only be awakened or restarted using the same number.

m. *p_text - pointer to the shareable portion of a process, if it exists.

2. U Vector (user.h)

The system associates 1024 bytes of storage with each user process, called the "u vector". This storage contains system per-process data and the system stack for this process. An important difference between user.h and proc.h is, proc.h is always core resident while user.h is

core resident only when the process it is associated with is core resident. The MUNIX scheduling algorithm does not use any elements from the u vector to make decisions.

B. SYSTEM FUNCTIONS PERTINENT TO SCHEDULING

The scheduling flow and basic system flow are shown in Figure 4 [7].

1. sched

This function is a process with an infinite loop initiated from function main. Sched is executed, put to sleep (see function sleep below), awakened (see function wakeup below), and executed again. Sched's main job is to swap processes in and out of core. It accomplishes this task using the following algorithm: If there are any processes in the swap file (out of core), find the one that has been there the longest and try to transfer it into core. If the swap file is empty, then go to sleep on the RUNOUT flag which indicates there are no processes on the swap file. When the longest waiting process has been found, sched tries up to three different ways to transfer it into core.

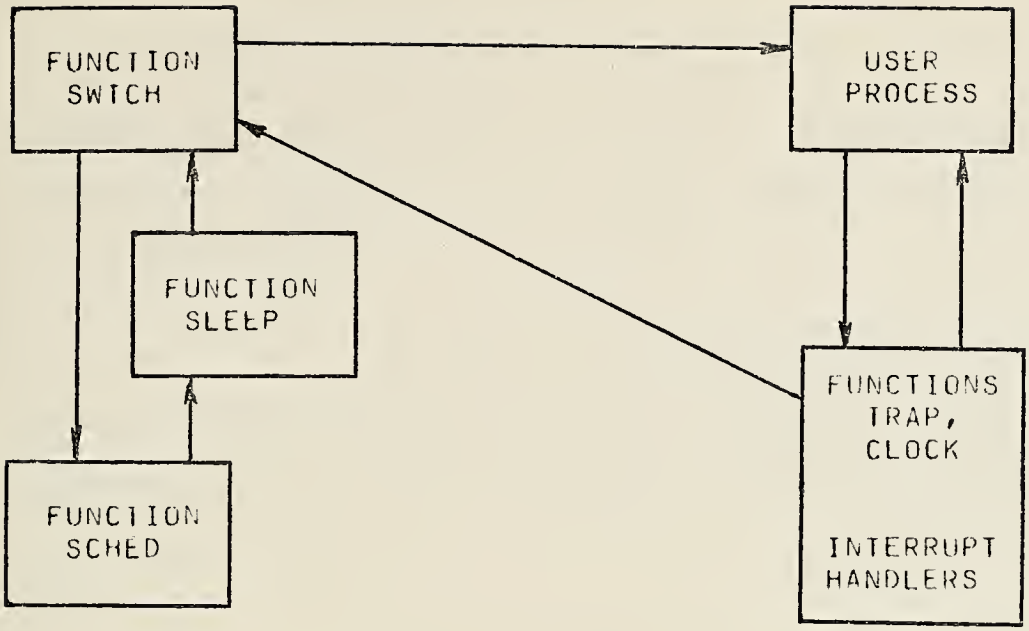


Figure 4. Scheduling Flow

a. If room exists in core then transfer the process in.

b. If room does not exist in core then sched looks for what it calls "easy core". Easy core is core that belongs to a process that is not a system process, not locked (eligible for transfer out), and waiting for some type of input or output. If easy core is found, then that process is transferred out and sched starts over by looking for the process that has been in the swap file the longest (this will be the same process that it found before), and continues with "a" above. Sched repeats this until either the process can be transferred in or no easy core is available.

c. If no easy core is available then sched makes two more checks to insure that the process is deserving enough to require another process to be transferred out.

(1) If the process has not been out of core for more than two seconds, sched goes to sleep on the RUNIN flag which indicates there is at least one process on the swap file.

(2) Find the process (it must be sleeping or ready to run, but not running) that has been in core the longest. If that process has not been in core at least two seconds then sched goes to sleep on the RUNIN flag. If it has been in more than two seconds, transfer it out and start over by looking for the process that has been out of core the longest and continuing with "a" above.

2. swtch

This function is invoked several places throughout MUNIX to accomplish the task of rescheduling the CPUs. Swtch searches the process table for the highest priority process that is in core and ready to run on the requesting CPU. If a process is found it is given the CPU, otherwise the CPU is put in an idle state. It stays in an idle state until started again by an interrupt.

3. sleep

This function is also invoked several places throughout MUNIX. It will change the process' status from ready to sleeping or waiting depending on the value of pri. If pri is less than zero, a signal cannot disturb the sleep, and the status is changed to SSLEEP. If pri is not less than zero, the status is changed to SWAIT, and the process may be disturbed by signals. Chan is an integer that represents the reason the process has been placed in a wait state (SWAIT or SSLEEP). After the process has been put in a wait state, sleep calls swtch to find another process to run.

4. wakeup

This function changes the status of all processes that have been put to sleep on chan from the wait state, to the SRUN state (ready to run). If any processes awakened are on the swap file and sched is sleeping on the RUNOUT flag, it is also awakened. When swtch is next called, sched will be scheduled to run (sched is the highest priority system process), and an attempt will be made to find core for all the processes just awakened by wakeup.

APPENDIX B: SYSTEM BENCHMARK

A. GENERAL

This appendix contains information concerning the benchmark program used for testing and evaluating scheduling changes in this thesis.

B. INDIVIDUAL PROCESSES

Times for all the individual processes can be found in Table VIII. The times used in Table VIII are from the "A" system (see section II.B.2). It is significant to note that the times on the "B" system are faster because there is approximately three times as much "user core" available. The benchmark consists of a series of eight processes (discussed below) executed from a command file.

1. chdir /usr/sys

Change the current working directory to the new one specified, in this case the new working directory is /usr/sys. This command does not create a new process, but is directly executed in the shell.

2. sh 1d&

Execute the command file ld. File ld loads a new operating system and places it in a file named "a.out".

3. chdir conf

See 1. above.

4. cc -c conf.c&

Compile without loading the C program named "conf.c"; this "program" consists of data statements, initialization, and no executable code. The compiled object code goes in a file named "conf.o".

5. chdir /usr/bench

See 1. above.

6. cc -O rftest.c&

Compile the C program "rftest.c" using the experimental object-code optimizer. The optimized object-code goes in a file named "a.out".

7. bas tower <towerin >/dev/null&

This is a compute bound process that has an input file named "towerin" and an output file named "/dev/null". The output file is a null device. Tower is an interpretive execution of a recursive solution to the towers of Braman (Hanoi) problem which represents tokens as double precision floating point numbers. Solution is for thirteen disks and three towers.

8. chdir /usr/sys/dmr

See 1. above.

9. cc -c -O tm.c&

Compile the C program named "tm.c" without loading it and with the experimental object-code optimizer. The resulting object-code goes in a file named "tm.o".

10. cp /munix /dev/null&

Copy the 34,800 byte file named "/munix" to the file named "/dev/null".

11. chdir /usr/sys

See 1. above.

12. sum /usr/sys/lib1 >/dev/null&

Compute the checksum of the 60,390 byte file named "/usr/sys/lib1" and output the number to the file named "/dev/null".

13. sum /usr/sys/lib2 >/dev/null&

See 12. above.

14. wait

Wait until all processes started with "&" have completed, and report any abnormal terminations. There is no measurable time associated with this command if it stands alone.

All the times used in Table VIII have come from the time command of UNIX [18]. Execution times (user and system) are determined by sampling the state of the system at a 60 hz rate (1/60 second). A counter is kept for each type of time. Note that "nm" means not measurable. It is significant to note that the execution time can depend on what kind of memory the process happens to occupy. The user time in MOS is approximately half of what it is in core [18]. This problem has been solved by running the benchmark program as a single user. This forces the same processes into the same type of core. The elapsed time (real) is accurate to the second, while the CPU times (user and system) are measured to the 60th of a second. It was found that the system times may vary by as much as 8.5 per cent, and the real time by as much as 8.3 per cent.

process	real	user	sys
1	nm	nm	nm
2	40	4.7	3.9
3	nm	nm	nm
24	21	1.0	1.9
5	nm	nm	nm
6	37	4.2	4.0
7	34	30.8	0.7
8	nm	nm	nm
9	41	10.9	3.4
10	5	0.0	0.6
11	nm	nm	nm
12	6	0.3	0.7
13	5	0.3	0.6
14	nm	nm	nm
sum (min)	3:09	52.2	15.8

Table VIII. Individual Process Times

C. BENCHMARK PROGRAM

The benchmark program consists of all the processes mentioned in section B. above in a multiprogrammed mix. Time for the multiprogrammed benchmark is:

```
real 7:08
user   46.6
sys    18.0
```

Table IX. benchmark Program Evaluation

The command sequence "time sh benchmark" is used to initiate processing of the benchmark.

APPENDIX C: NON-ADAPTIVE SCHEDULING CHANGES

A. GENERAL

This appendix contains detailed information concerning the non-adaptive changes made to the MUNIX (based on UNIX Version 5) scheduling algorithm.

B. MAXIMUM NUMBER OF PROCESSES (NPROC)

1.Change

NPROC was a constant defined in param.h to be fifty. That means there can be no more than fifty processes in the system at any one time. Twelve system functions used NPROC for searching the process table. For example: if function swtch was looking for the highest priority process, it looked at all entries in the process table. Normally this would not be considered wasteful, but the process table is very seldom, if ever, completely full. This means there is time being wasted if the process table does not hold fifty processes. Since processes are entered into the process table at the first available space, a counter could be used to hold the maximum number of processes in the process

table. Time could be saved by searching the process table from the beginning to the counter. This change was made as follows:

1. A new integer variable, nproc (lower case), was placed in proc.h to keep track of the last process in the process table. The twelve functions that used NPROC now use nproc.

2. Two lines of code have been added to function newproc (in program slp.c). The code insures that nproc is incremented when necessary.

3. Two lines of code have been added to function wait (in program sys1.c) to insure that when the last process in the process table terminates, nproc is decremented. It is not sufficent to decrement nproc by one in all cases. Example: The process table could have the first nine process blocks allocated, a new process enters and takes block ten, process eight and nine terminate, then process ten terminates. If nproc was decremented by one, then all searches would look at blocks eight and nine unnecessarily.

2. Evaluation

The benchmark program (see Appendix b) was run against the scheduling algorithm before and after this

change. Four runs were made, two with a drum being used for /TMP files (temp files), and two without. The results are listed in tables III and IV. Real, user, and system times are shown in minutes and seconds. Appendix B explains how the system calculates these times and estimates their accuracy. This testing was accomplished on the "B" system (see section II.B.3.).

BEFORE CHANGES		AFTER CHANGES	
real	6:02	real	6:00
user	2:12	user	2:00
sys	:42	sys	:41

Table III. NPROC Change Evaluation with No Drum

Before Changes		After Changes	
real	4:49	real	4:38
user	1:58	user	1:48
sys	:41	sys	:42

Table IV. NPROC Change Evaluation with Drum

C. LOOPING IN FUNCTION SCHED

1. Change

As described in section B.1.b and B.1.c.(2) of Appendix A, sched unnecessarily loops to a point that is

repetitive. The two loops were changed as follows:

1. A label, "findsp" (find space for process), was inserted where sched starts looking for core for the process it just found (the process that has been out of core the longest). When easy core has been found, instead of branching back to look for the process that has been out of core the longest, sched branches to findsp.

2. A pointer, "p2", was added to the declarations of sched. The pointer p2 was substituted for p1 in the first two instances after no easy core is found. This leaves p1 pointing to the process that has been out of core the longest, giving no need to search for that process again.

2. Evaluation

The benchmark program (see appendix B) was run before and after the changes. Several runs were made because the statistics showed no significant savings (see Table V). Testing was accomplished on the "A" system.

Before Changes	After Changes
real 7:08	real 7:08
user :46.6	user :46.6
sys :18.0	sys :17.8

Table V. Looping Change Evaluation

D. SIZE CHECK

1. Change

In two separate places, function sched swapped a process out of core without giving any consideration as to whether that action created enough room for the incoming process. Many times two or three processes were swapped out of core when only one was necessary. This creates a large overhead in swapping. A two pass check was installed to circumvent this problem.

1. First pass - Check to see that the process being swapped out of core is as large or larger than the one being swapped in, if not, do not swap it out.

2. Second pass - If the first pass fails to create enough room for the incoming process, swap eligible processes out until enough room exits.

This change was implemented using a first/second pass indicator (fspass) having the value of zero for the first pass and one for the second pass. This indicator was "or"ed with the size check thereby using the same code for both passes.

2. Evaluation

Several runs were made with the benchmark program on the "A" system because of the 26 percent savings realized in real time (see Table VI). This change was also tested on the "B" system, but a savings of only 6 percent was found there. The difference in savings is explained by the significant difference in available memory for each system.

BEFORE CHANGES	AFTER CHANGES
real 7:08	real 5:18
user :46.6	user :45.3
sys :18.0	sys :17.9

Table VI. Size Check Change Evaluation

APPENDIX D: PRIORITY CALCULATION SUBROUTINE

```
/*
 * This subroutine was written by Ronald E. Joy and
 * used for an adaptive scheduler in November of 1975.
 */


include "../param.h"
include "../proc.h"


/*
 * If the following variables are needed in any
 * other program, include schpri.h
 */


    int tchg1     4;    // slope 1 changes at this
                        // time (seconds).
    int slop1     2;    // number of bits to shift
                        // left = *4 (slope 1).
    int slop2     0;    // number of bits to shift
                        // left = *1 (slope 2).
    int chgt1    16;    // if tchgt1 or slop1 are
                        // changed, chgt1 must be
                        // changed also.
                        // chgt1 = tchg1 << slop1
    int bgpri   300;    // back ground priority
    int minpri -300;    // lowest priority (value)
    int maxpri  300;    // max priority (value)
    int mtpri   250;    // max time priority
    int mxtime  540;    // max time = 9 min (sec.)
    int maxres 7200;    // max resource units. this
                        // value is = 2 minutes.

/*
 * The following code is used to set a users priority
 * between -300 and 300. A value of 0 is the least
 * critical priority, with -300 being the most
 * critical. Any value over 0 is non-critical. A
 * process is critical if it has not received as
 * many resource units as dictated by the policy
 * function.
 */
```

```
schpri(nrp) // schedule priority
struct proc *nrp;    // pointer to process that needs
                     // a priority calculated.
{
    register struct proc *rp;   // process pointer
    register int pri;   // calculated priority
    register int resr;  // resource units received
    int time;   // current time of this process

    rp = nrp;
    time = rp->p_time;
    resr = rp->p_resr;
    if (rp->p_flag&PSTM || rp->p_flag&IPWAIT)
    // if this process is already a back ground
    // process or it is waiting for terminal I/O
        pri = bgpri;     // priority = back ground
    else {
        if (resr < 0)   // has the resource count
        // gone over 32767. there is no chance
        // of this happening with mxtime set to
        // its current value.
            {rp->p_flag =| PSTM; pri = bgpri;}
            // set PSTM flag and priority to bg
        else {
            if (time >= mxtime) // if process has
            // been alive longer than mxtime
                if (resr > maxres)  // if resource
                // count greater than maxres
                    {rp->p_flag =| PSTM; pri = bgpri;}
                        // set PSTM and pri to bg
                else
                    pri = mtpri;    // max time priority
            else {
                if (time < tchg1)   // if process
                // has been alive less than tchg1
                    pri = resr - (time << slop1);
                else
                    pri = resr - chot1 -
                ((time -tchg1) << slop2);
                if (pri > maxpri)   // if priority
                // is too large
                    pri = maxpri;   // fix it
                else
                    if (pri < minpri)   // too small
                        pri = minpri;   // fix it
                }
            }
        }
    return (pri);   // return priority to caller
}
```

BIBLIOGRAPHY

[1] Bell Laboratories, "C - Reference Manual," 1972.

[2] Bernstein, A.J., and Sharp, J.C., "A Policy-Driven Scheduler for a Time-Sharing System," Communications of the ACM, V.14, No.2, p.74-78, February 1971.

[3] Coffman, E.G., Jr., and Kleinrock, L., "Computer Scheduling Methods and their Countermeasures," Proc AFIPS 1968 SJCC, V.32, p.11-21, 1968.

[4] Digital Equipment Corporation, "PDP 11/45 Processor Handbook," 1974-75.

[5] Digital Equipment Corporation, "Peripherals Handbook," 1973-74.

[6] Doherty, W.J., "Scheduling TSS/360 for Responsiveness," Proc AFIPS 1970 FJCC, V.37, p.97-111, 1970.

[7] Hawley, J.A., III, and Meyer, W., "MUNIX, A Multiprocessing Version of UNIX," Master's Thesis, Naval Postgraduate School, Monterey, 1975.

[8] Hellerman, H., "Some Principles of Time-Sharing Scheduler Strategies," IBM Systems Journal, V.8, No.2, p.94-117, 1969.

[9] IBM Research Report RC 3672, "The Effects of Adaptive Reflective Scheduling," by W.J. Doherty, September 1, 1971.

[10] Kernighan, B.W., "Programming in C - A Tutorial," Bell Laboratories, 1974.

[11] Kleinrock, L., "A Continuum of Time-Sharing Scheduling Algorithms," Proc AFIPS 1970 SJCC, V.36, p.453-458, 1970.

[12] Kral, T.C., "A Process Controller for a Hierarchical Process Structured Operating System," Master's Thesis, Naval Postgraduate School, Monterey, 1975.

[13] Northouse, R.A., and Fu, K.S., "Dynamic Scheduling of Large Digital Computer Systems Using Adaptive Control and Clustering Techniques," IEEE Transactions on Systems, Man, and Cybernetics, V. SMC-3, No.3, p.225-234, May 1973.

[14] Raetz, G.M., "Adaptive Memory Management in a Paging Environment," Master's Thesis, Naval Postgraduate School, Monterey, 1973.

[15] Ritchie, D.M., and Thompson, K., "The UNIX Time-Sharing System," Communications of the ACM, V.17, No.7, p.365-375, July, 1974.

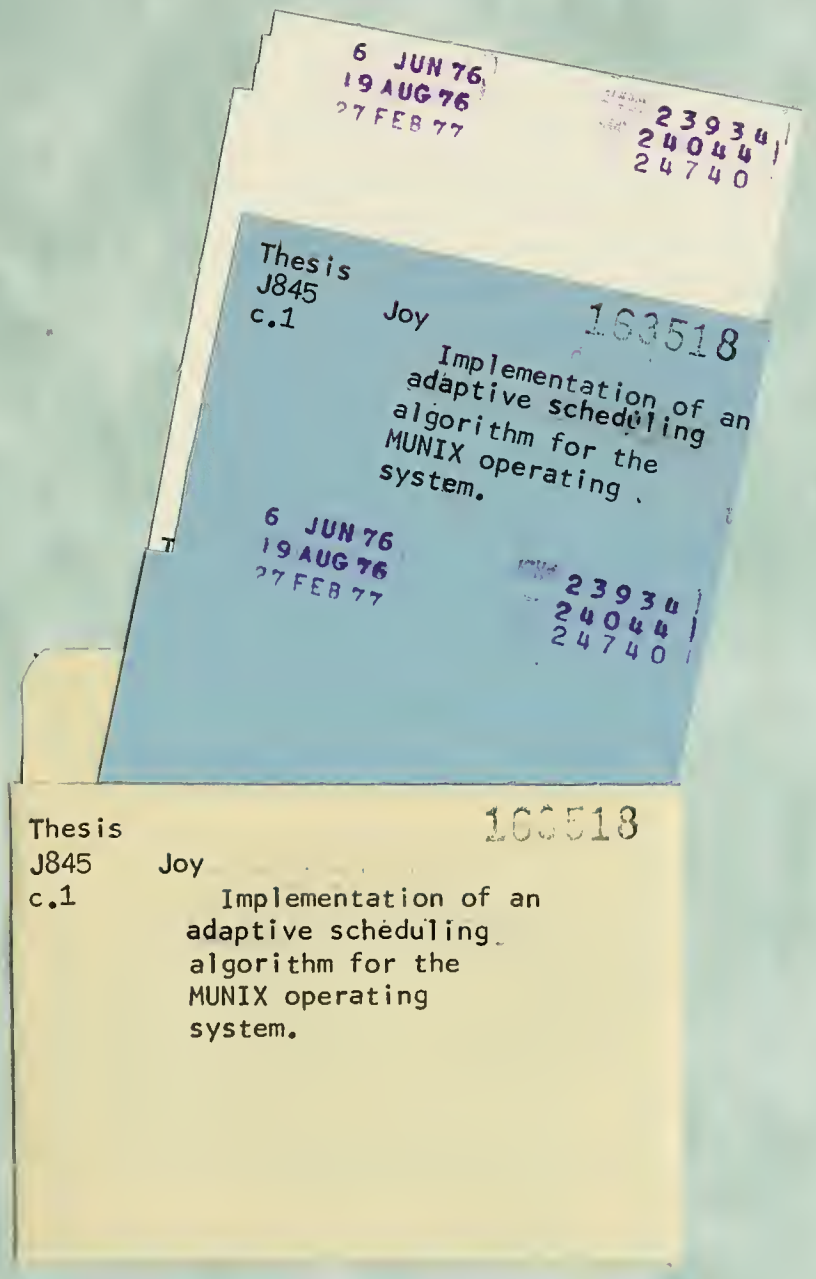
[16] Sharp, J.C., and Roberts, J.N., "An Adaptive Policy Driven Scheduler," ACM Performance Evaluation Review, V.3, No.4, p.199-208, December 1974.

[17] Shinabarger, J.A., "Adaptive Scheduling in a Multiprocessing Environment," Master's Thesis, Naval Postgraduate School, Monterey, 1973.

[18] Thompson, K., and Ritchie, D.M., "UNIX Programmer's Manual," 5th ed., Bell Laboratories, 1974.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Documentation Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0212 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 72 Computer Science Group Naval Postgraduate School Monterey, California 93940	1
4.	Assistant Professor Gerald L. Barksdale, Jr. Code 72Ba Computer Science Group Naval Postgraduate School Monterey, California 93940	1
5.	Assistant Professor Belton E. Allen, Code 72An Computer Science Group Naval Postgraduate School Monterey, California 93940	1
6.	Air Force Institute of Technology AFIT/CIDD Wright-Patterson Air Force Base Ohio 45433	1
7.	Captain Ronald E. Joy Department of Computer Science United States Air Force Academy Colorado 80840	1



Thesis
J845
c.1
Joy
163518
Implementation of an adaptive scheduling algorithm for the MUNIX operating system.